Analysis of Data Reuse in Task Parallel Runtimes

Miquel Pericàs[⋆], Abdelhalim Amer[⋆],
Kenjiro Taura[†] and Satoshi Matsuoka[⋆]

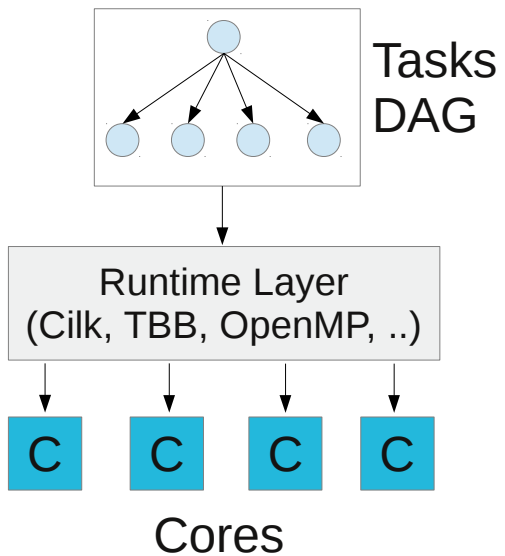
[⋆]Tokyo Institute of Technology
[†]The University of Tokyo

Table of Contents

Table of Contents

Task Parallel Programming Models

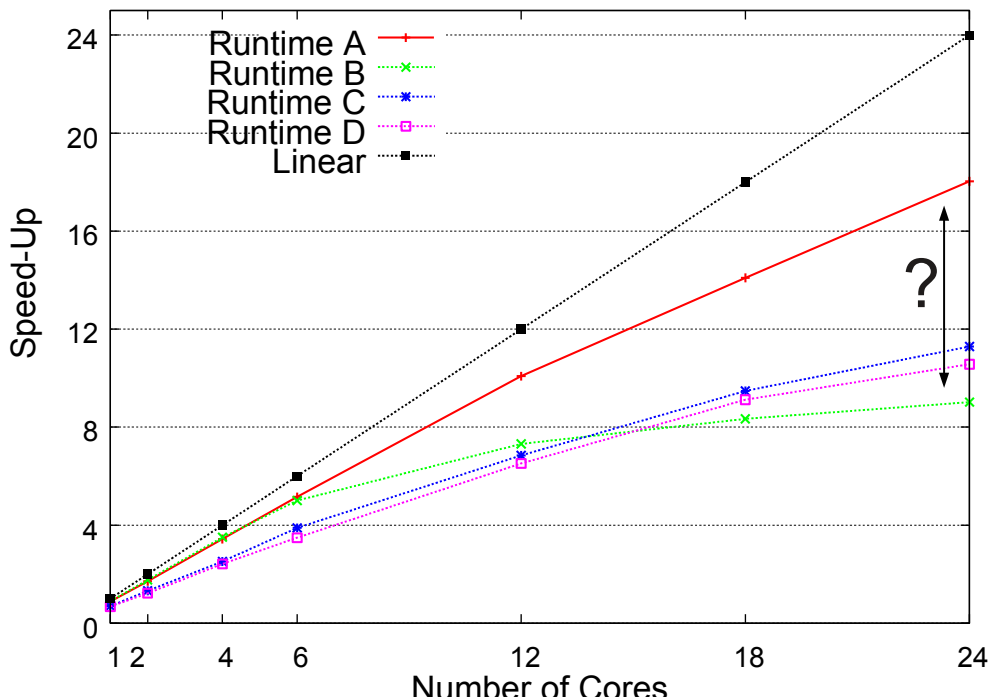
- Task-parallel programming models are popular tools for multicore programming
- They are general, simple and can be implemented efficiently



- Task-parallel runtimes manage assignation of tasks to cores, allowing programmers to write cleaner code

Performance of Runtime Systems

- Runtime schedulers implement heuristics to maximize parallelism and optimize resource sharing
- Performance can depend considerably on such heuristics, degradation often occurs without any obvious reason



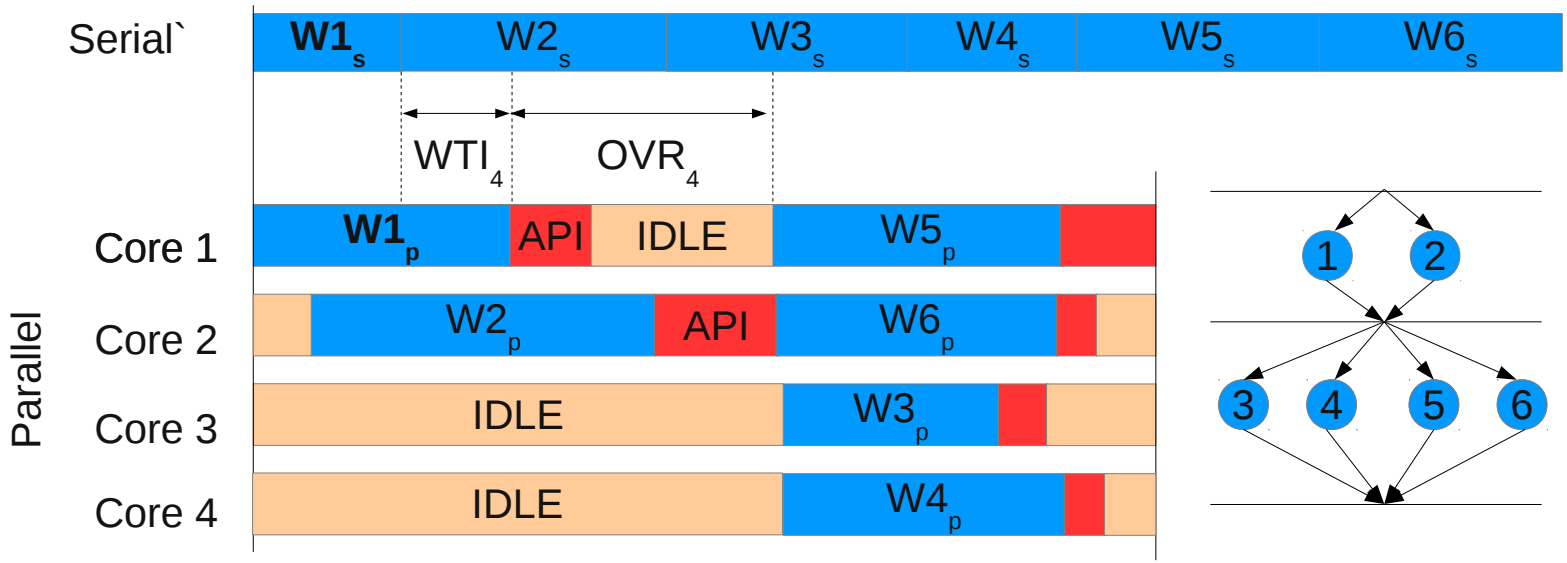
Scalability of task parallel applications

Why do task parallel codes not scale linearly?

- **Runtime Overheads**: execution cycles inside API calls
- **Parallel Idleness**: lost cycles due to load imbalance and lack of parallelism
- **Resource Sharing**: additional cycles due to contention or destructive sharing $\rightarrow$ work time inflation (WTI)

Quantifying Parallelization Stretch

- OVR_N = *Non-work Overheads* at N cores (API + IDLE time)
- WTI_N = *Work Time Inflation* at N cores



Parallel Stretch

$$T_{par} = \frac{T_{ser}}{N} \times \text{WTI}_N \times \text{OVR}_N \rightarrow \text{Speed-Up}_N = \frac{N}{\text{OVR}_N \times \text{WTI}_N}$$

Table of Contents

Case Study: Matmul and FMM

Matrix Multiplication (C = A x B)

- **Input Size**: 4096×4096 elements
- **Task Inputs/Outputs**: 2D submatrices
- **Average task size**[1]: 17 µs

Fast Multipole Method: Tree Traversal[2]

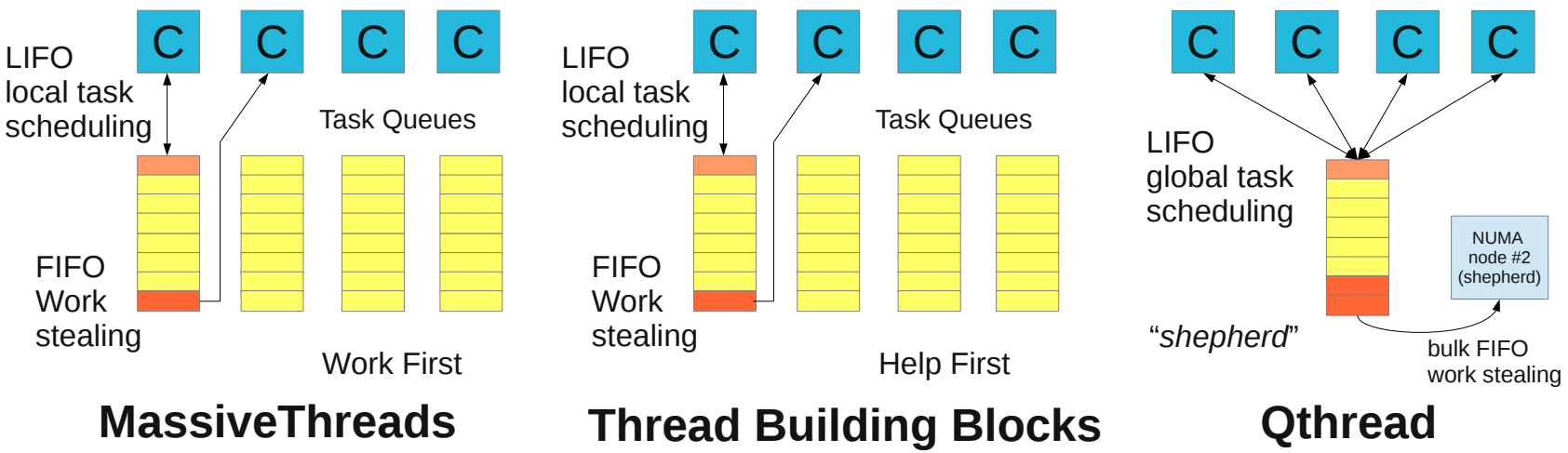
- **Input Size**: 1 million particles (Plummer)
- **Task Inputs/Outputs**: octree cells (multipoles and vectors of bodies)
- **Average task size**: 3.25 µs

[1] measured on Intel Xeon E7-4807 at 1.86GHz

[2] `https://bitbucket.org/rioyokota/exafmm-dev`

Case Study: three runtimes

- **MassiveThreads**: Cilk-like runtime with random work stealer and work-first policy.
- **Threading Building Blocks**: C++ template based runtime with random work stealer and help-first policy.
- **Qthread**: Locality-aware runtime with shared task queue. A set of workers are grouped in a *shepherd*. Bulk work stealing across shepherds (50% of victim's tasks). Help-first policy.

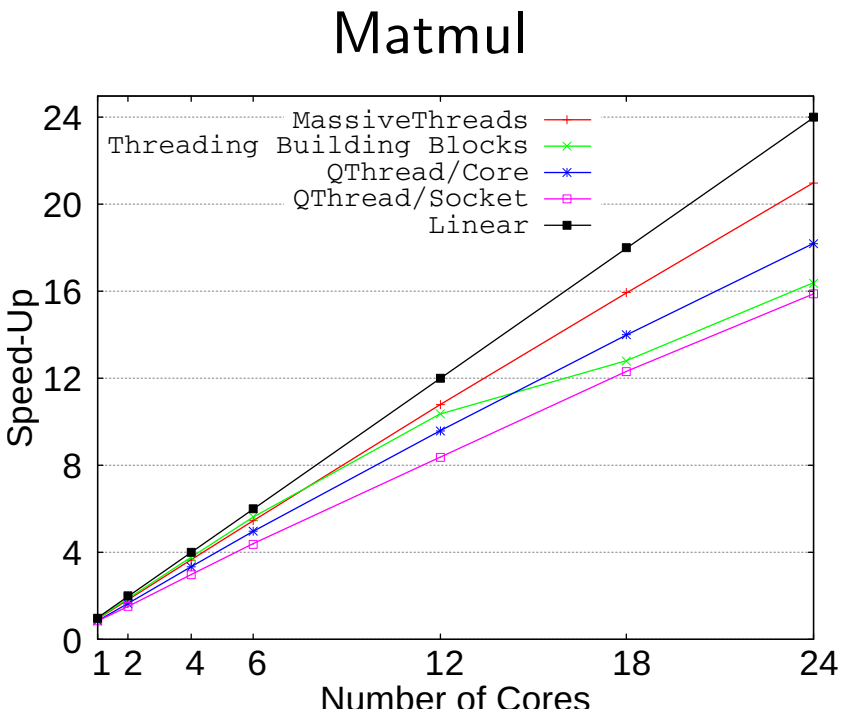


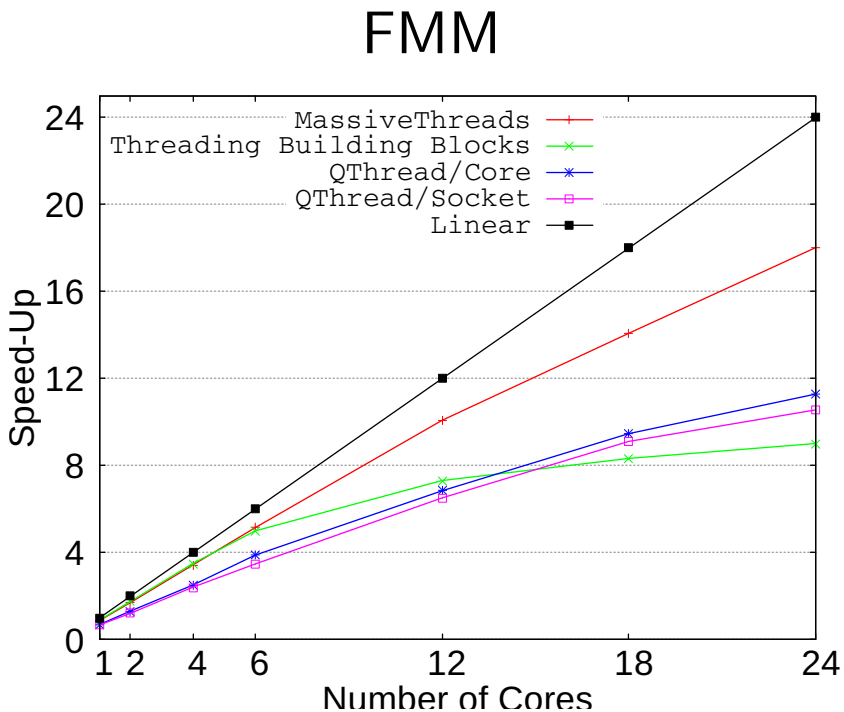
Experimental Setup

- Experimental platform is a 4-socket Intel Xeon E7- 4807 (Westmere) machine with 6 cores per die (1.87GHz) and 18MB of LLC.
- We specify the same subset of cores for every experiment
- The following runtime configurations are used:

Runtime	Task Creation	Work Stealing	Task Queue
MTH	Work-First	Random / 1 task	Core/LIFO
TBB	Help-First	Random / 1 task	Core/LIFO
QTH/Core	Help-First	Random / Bulk	Core/LIFO
QTH/Socket	Help-First	Random / Bulk	Socket/LIFO

Speed-Ups for Matmul and FMM

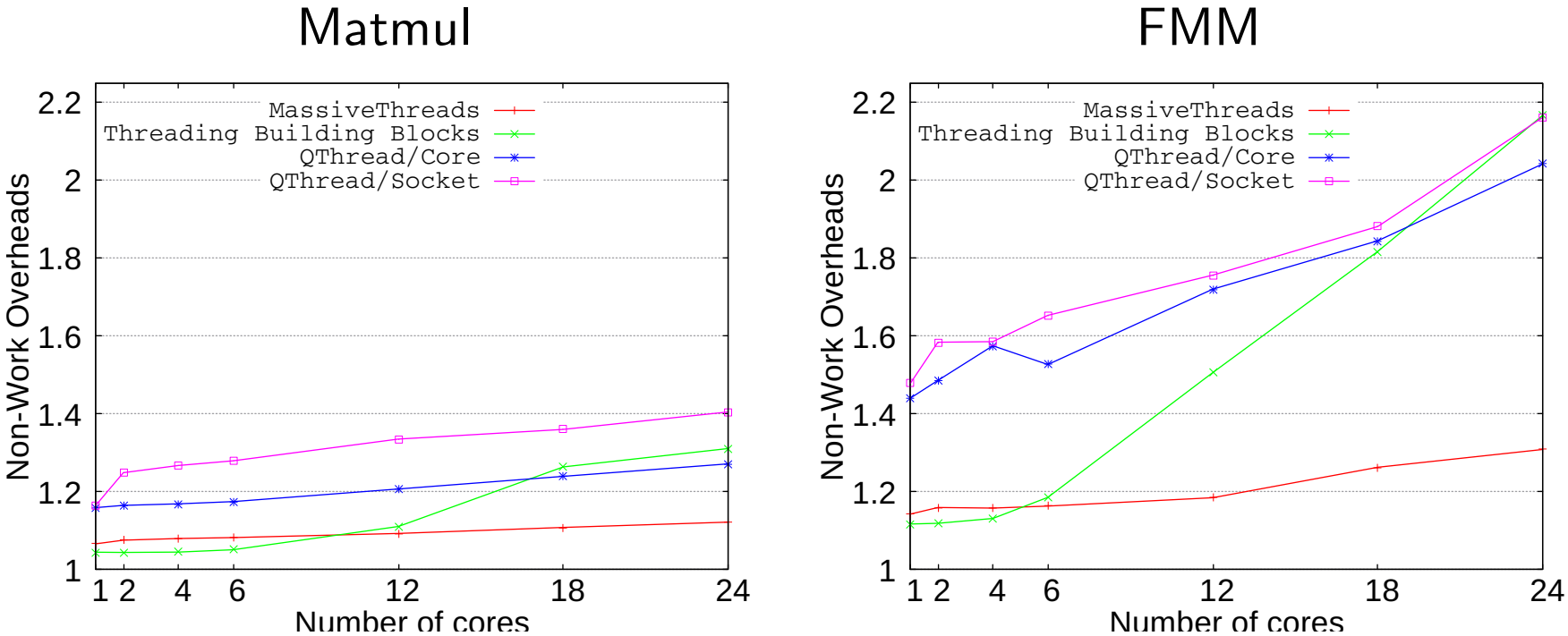




Performance Variation at 24 Cores:

- Matmul: 16×–21× (MTH best, QTH/Socket worst)
- FMM: 9×–18× (MTH best, TBB worst)

Overheads (OVR_N) for Matmul and FMM



Overheads are obtained by measuring the time cores spend outside of work kernels. At 24 cores:

- Matmul: 1.1×–1.4× (MTH best; QTH/Socket worst)
- FMM: 1.3×–2.2× (MTH best; TBB and QTH/Socket worst)

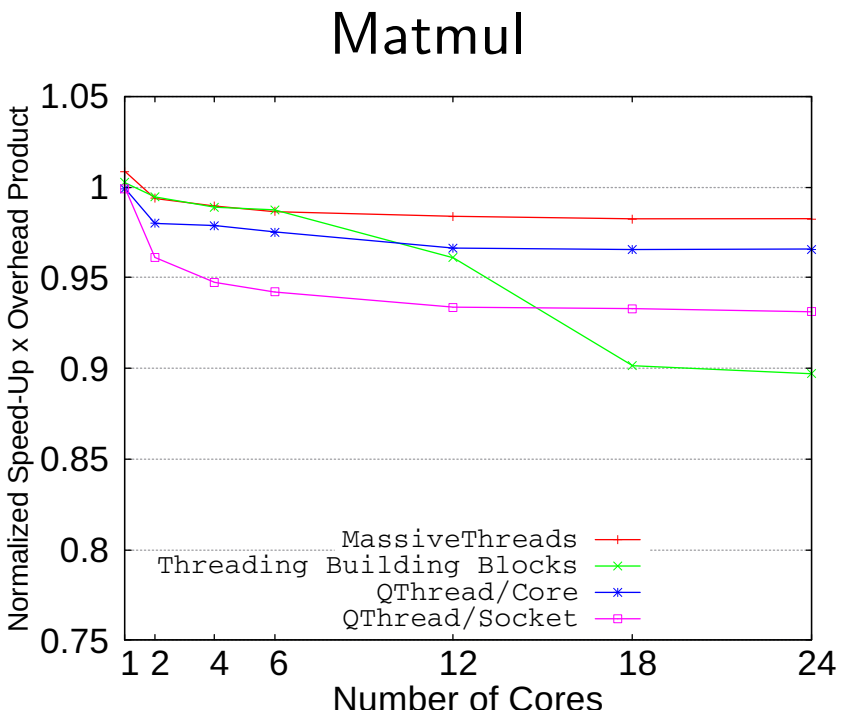
Do overheads alone explain performance?

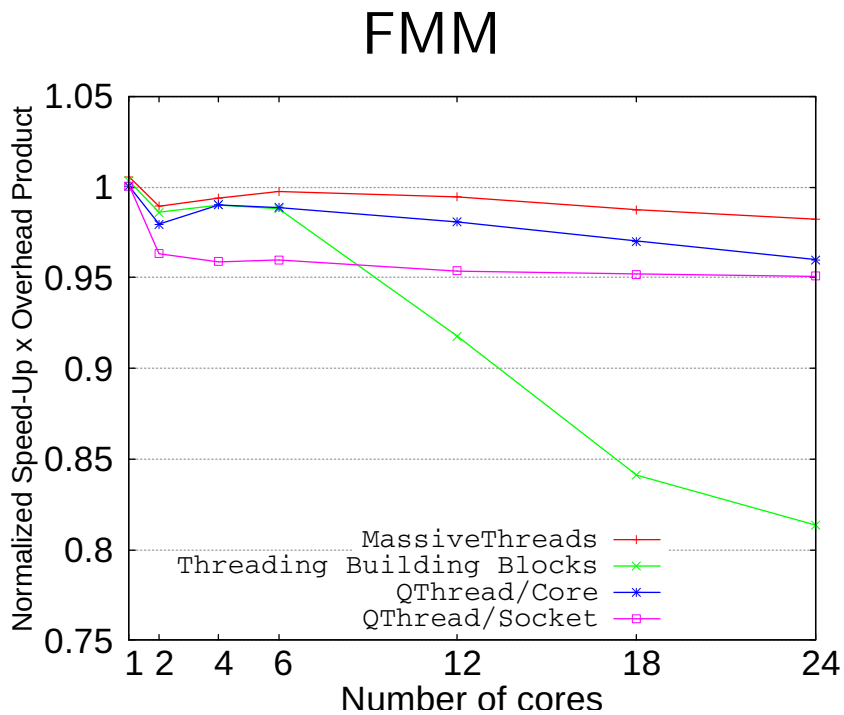
Normalized speed-up overhead product

$$\text{Speed-Up}_N = \frac{\text{N}}{\text{OVR}_N \times \text{WTI}_N} \rightarrow \frac{\text{Speed-Up}_N \times \text{OVR}_N}{\text{N}} = \frac{1}{\text{WTI}_N}$$

- The normalized speed-up overhead product is a measure of performance loss due to resource sharing
- A value of 1.0 means no work time inflation is occurring

Normalized speed-up overhead product





Speed-up degradation due to resource contention

- Matmul: 2%–10% (MTH best; TBB worst)
- FMM: 2%–18% (MTH best; TBB worst)
- Reason? cache effects due to different orders of tasks

Performance bottlenecks analysis

- Overheads can be studied with a variety of tools
  - **Sampling-based**: `perf`[1], HPCToolkit[2], extrae[3], etc
  - **Tracing-based**: vampirtrace[4], TAU[5], extrae, etc
  - **Runtime library support**
- How can we analyze the impact of different runtime schedulers on data locality?

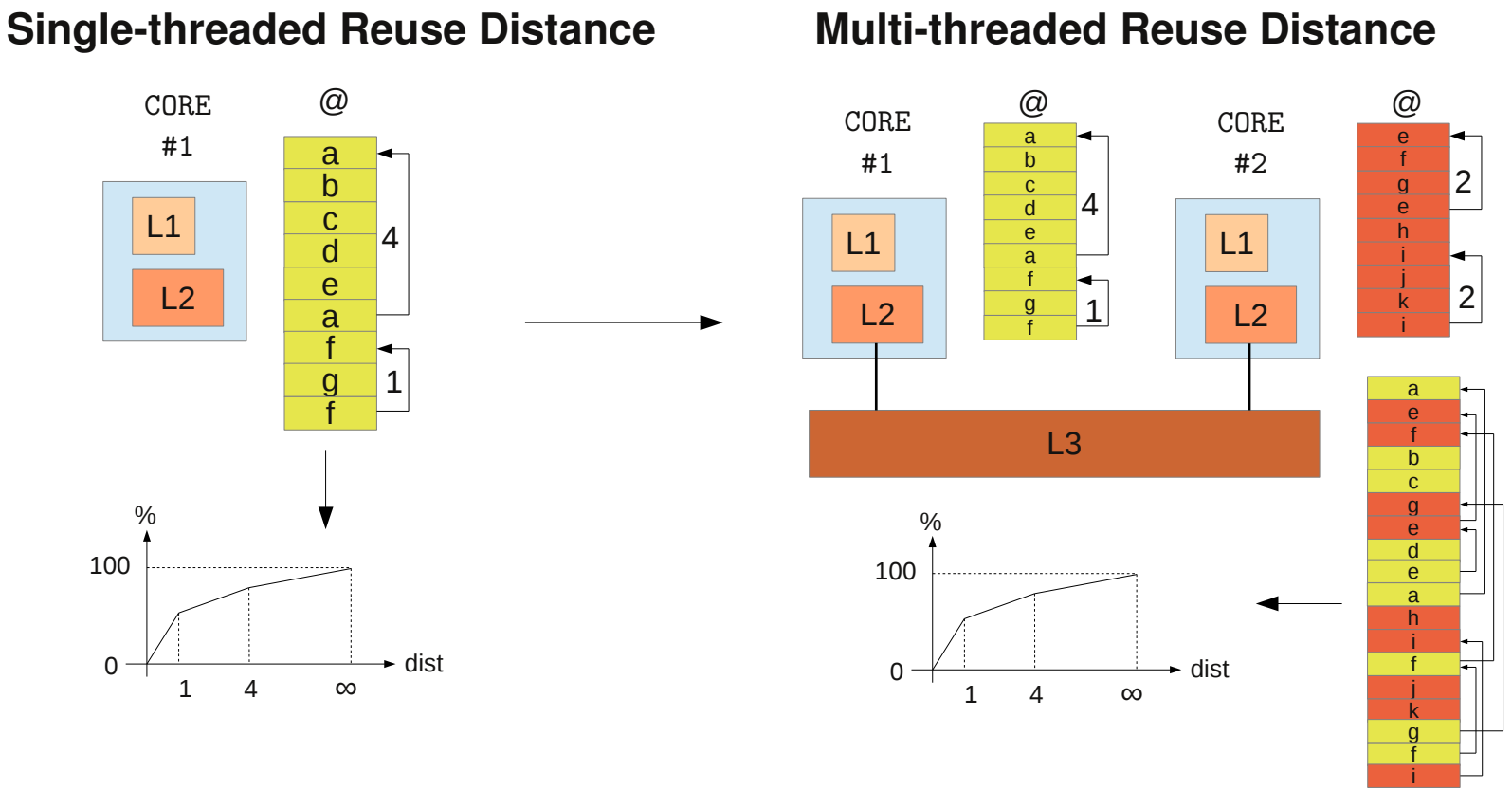
→ **Proposal**: use the reuse distance to evaluate cache performance

---

[1] `https://perf.wiki.kernel.org`
[2] `http://hpctoolkit.org/`
[3] `http://www.bsc.es/computer-sciences/performance-tools/paraver`
[4] `http://www.vampir.eu`
[5] `http://tau.uoregon.edu`

Table of Contents

Multicore-aware Reuse Distance



- Generation of full address traces is too intrusive
  $\rightarrow$ changes task schedules
- Computing the reuse distance is expensive

Lightweight data tracing

We make several assumptions to reduce the cost of the metric

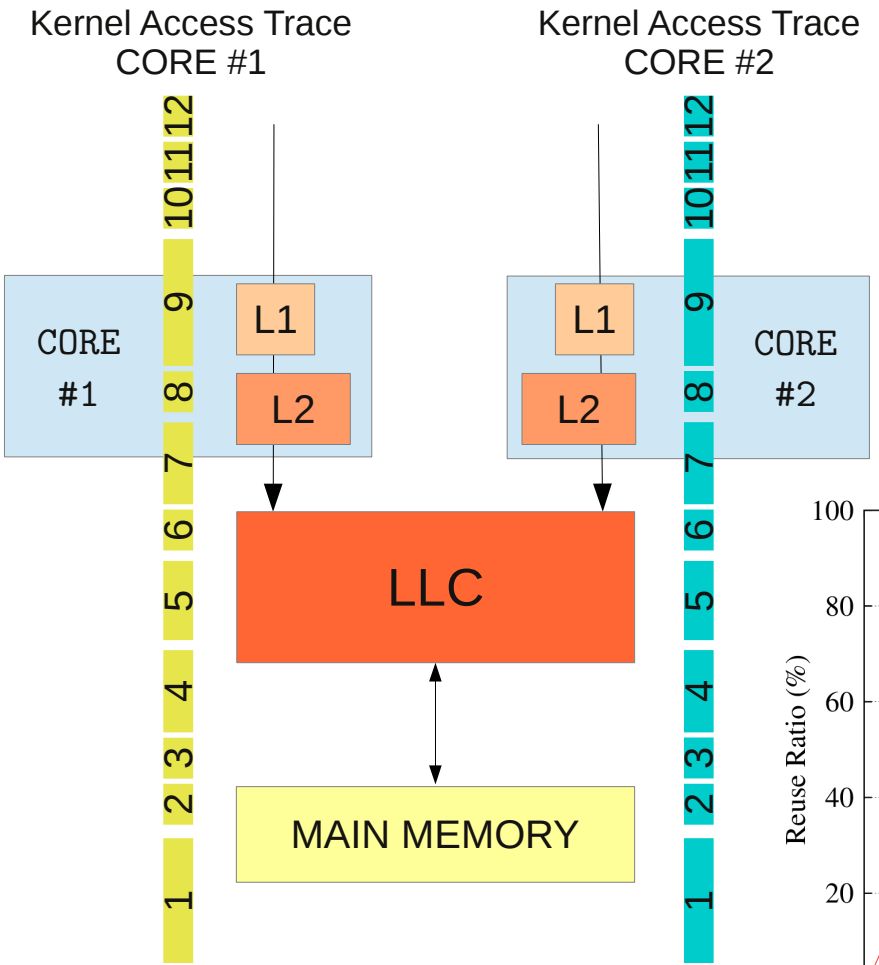
- Cache performance is dominated by global (shared) data

  $\rightarrow$ short lived stack variables are not tracked. Only the kernel inputs/outputs are recorded.
- Performance is dominated by last level cache misses

  $\rightarrow$ we interleave the address streams of all threads and compute the reuse distance histogram
- For large reuse distances individual LD/ST tracking is not needed

  $\rightarrow$ we record kernel inputs at bulk (`timestamp, address, size`)

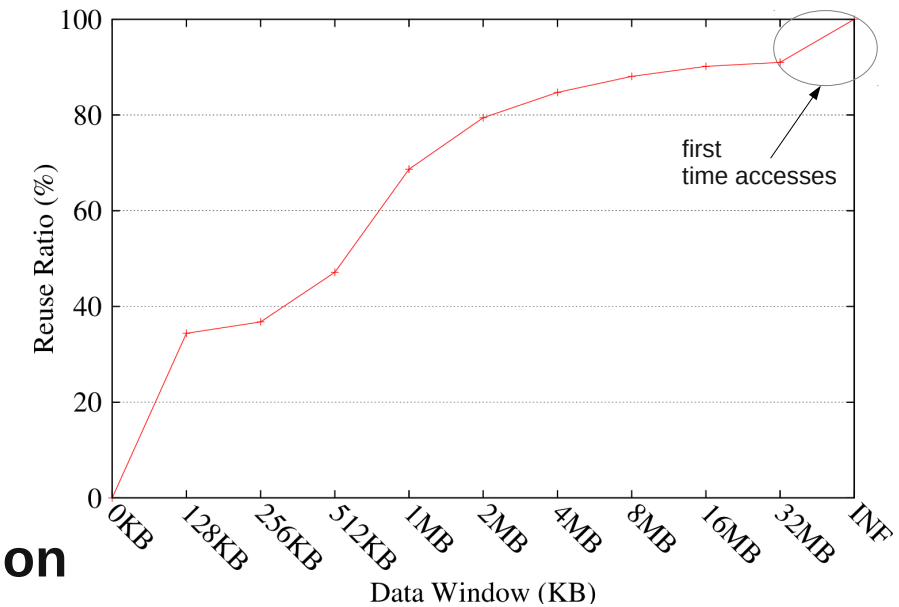
Kernel Reuse Distance (KRD)



1) Kernel Data Trace Generation

2) Merging/Synchronization

3) Histogram Generation



Kernel Reuse Distance: Application

Kernel Reuse Distance (KRD)

KRD provides an *intuitive* measure of data reuse quality. We want to make quick assessments on reuse, comparing the performance of different schedulers

Table of Contents

Instrumentation

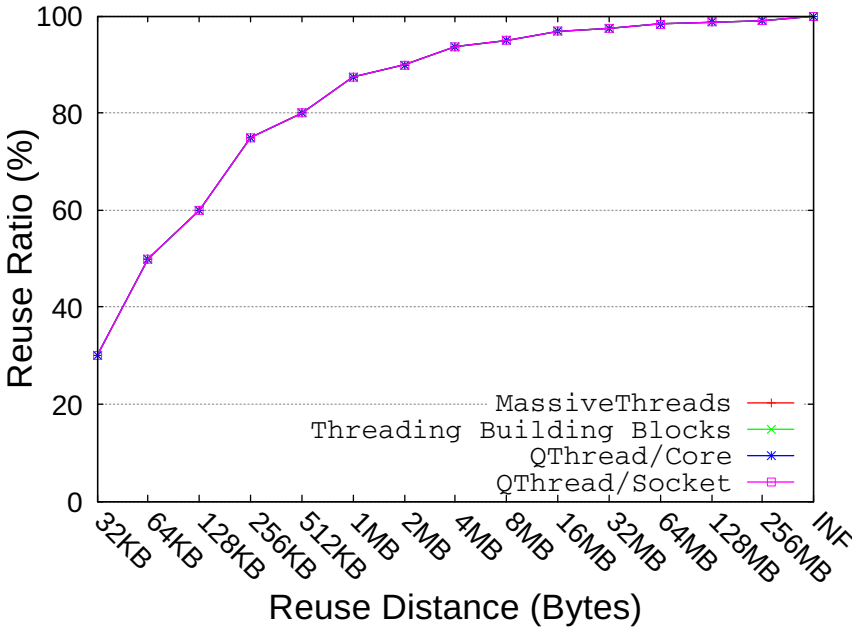
- We record submatrices for `matmul`, and multipoles and body arrays for `FMM`
- Total overhead below 5% for FMM and below 1% for Matmul
- As memory traces record data regions, histogram generation is much faster

KRD histograms and runtime schedulers

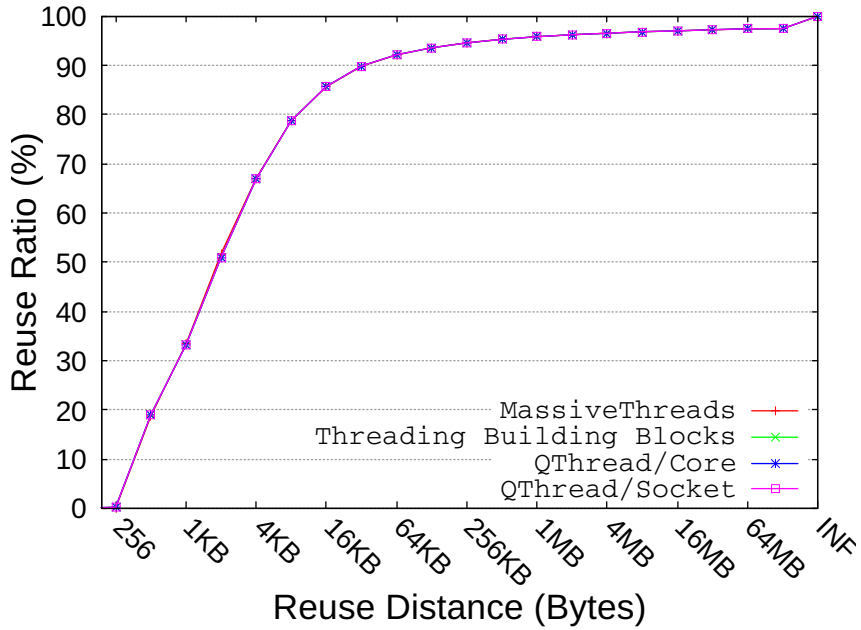
- We first analyze the correlation of different schedulers and the KRD metric:
- Four schedulers
  - MassiveThreads, TBB, Qthread/Core and Qthread/Socket
- Three system configurations:
  - 1 core
  - 1 socket (6 cores)
  - 4 sockets (24 cores)

Single Core Kernel Reuse Distance (KRD-1)

Matmul



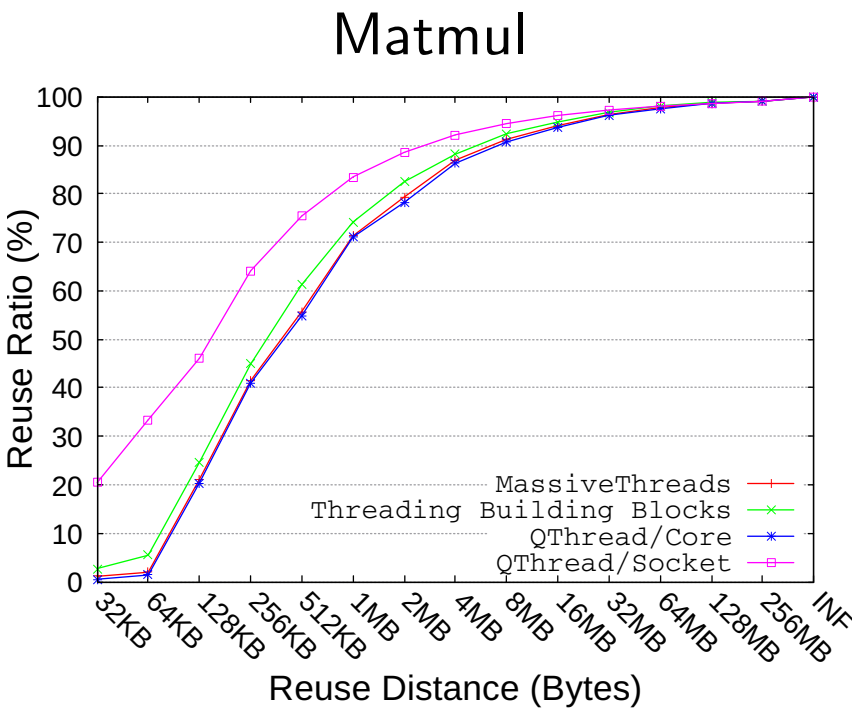
FMM

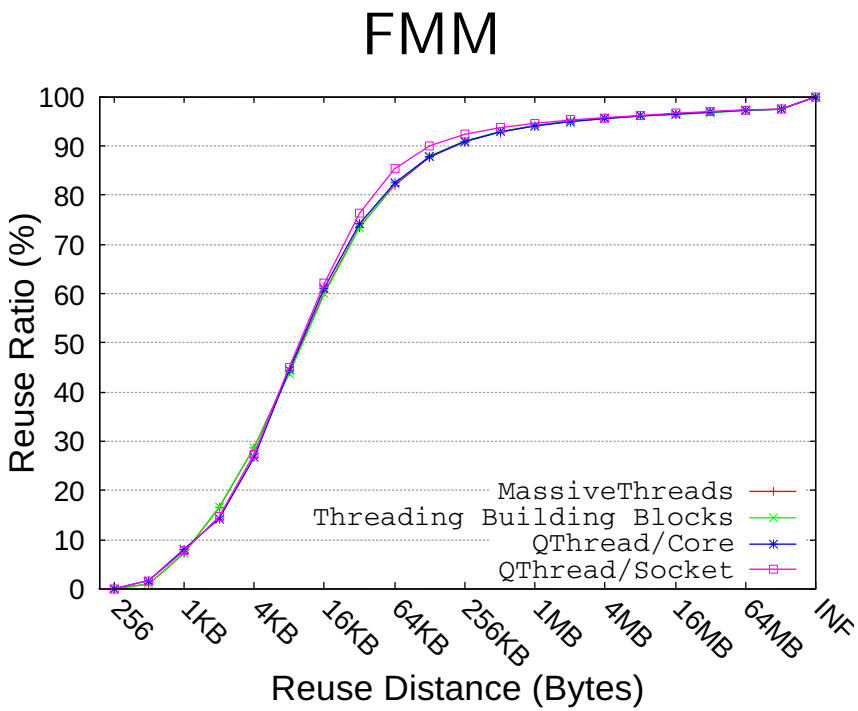


Almost no variations between histograms:

- In the abscence of work steals order is only determined by Work-First or Help-First
- Matmul kernel order is independent of spawn policy. FMM is sensitive, but differences are still minimal

Single Socket / 6 Core Kernel Reuse Distance (KRD-6)

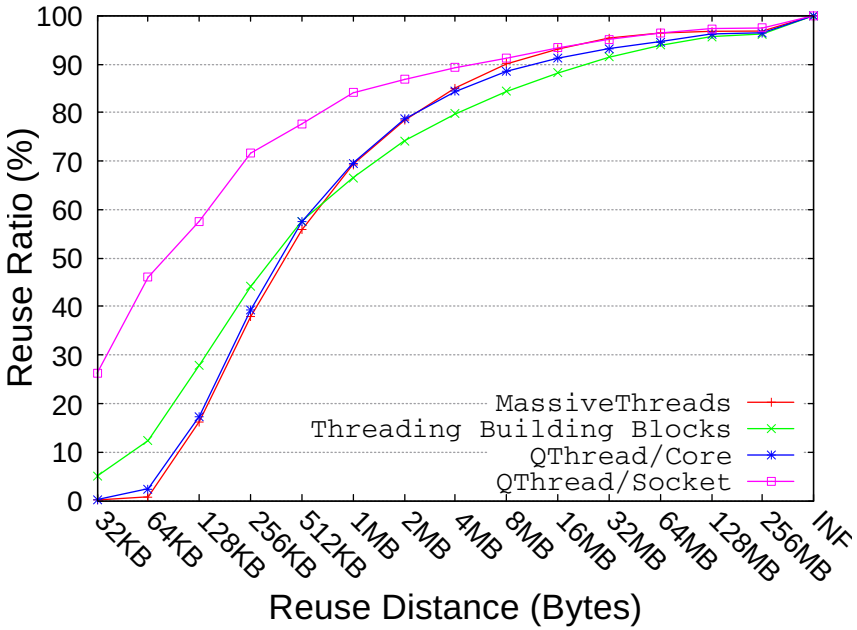




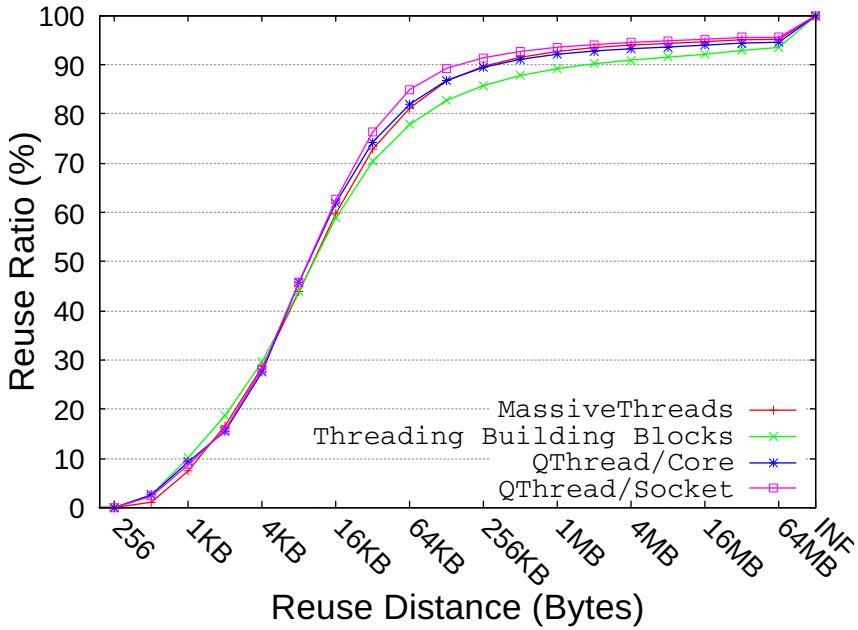
- QTH/Socket shared queue improves temporal locality
- Other schedulers almost no difference. TBB slightly better
- Differences in FMM are much smaller

Four Sockets / 24 Core Kernel Reuse Distance (KRD-24)

Matmul



FMM



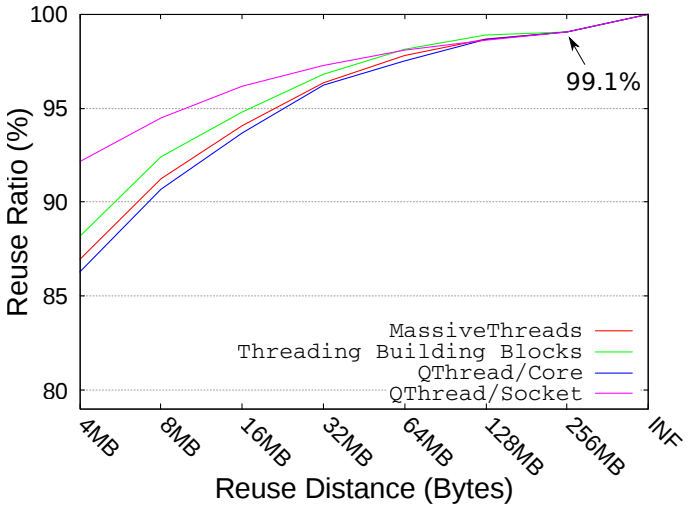
- Differences in distant reuses grow
- QTH/Socket shared queue also improves temporal locality with multiple sockets
- TBB suffers in the context of multiple sockets

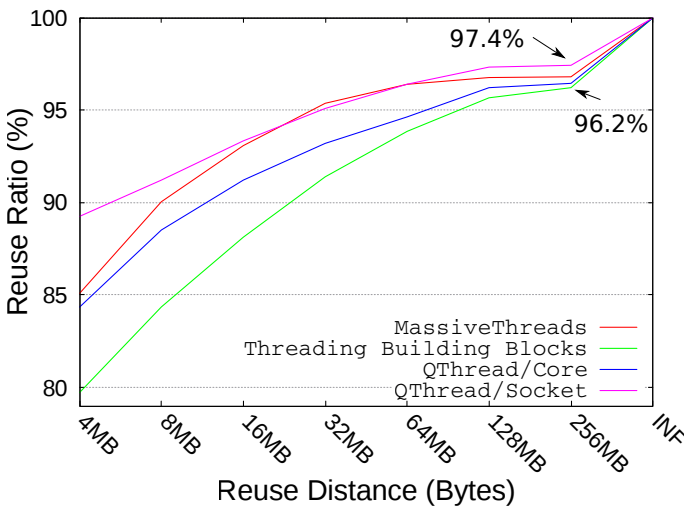
Impact of Multiple Sockets on Cold Accesses

1 socket

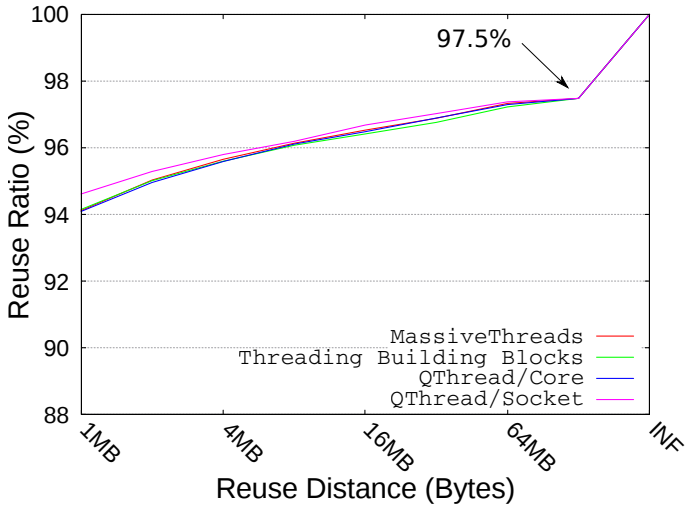
4 sockets

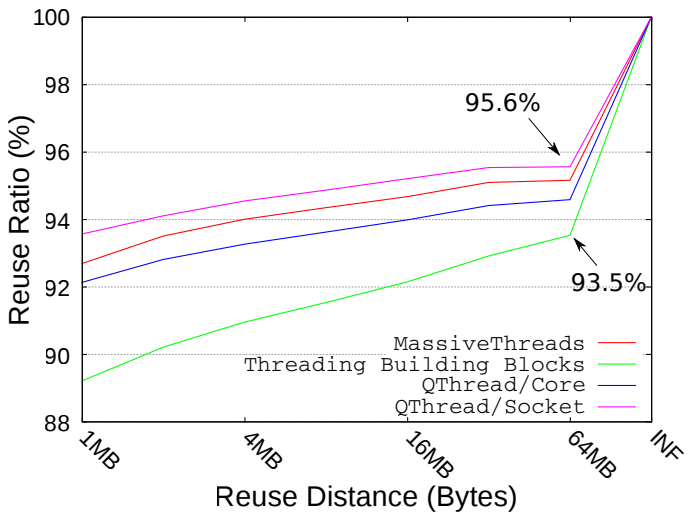
Matmul





FMM



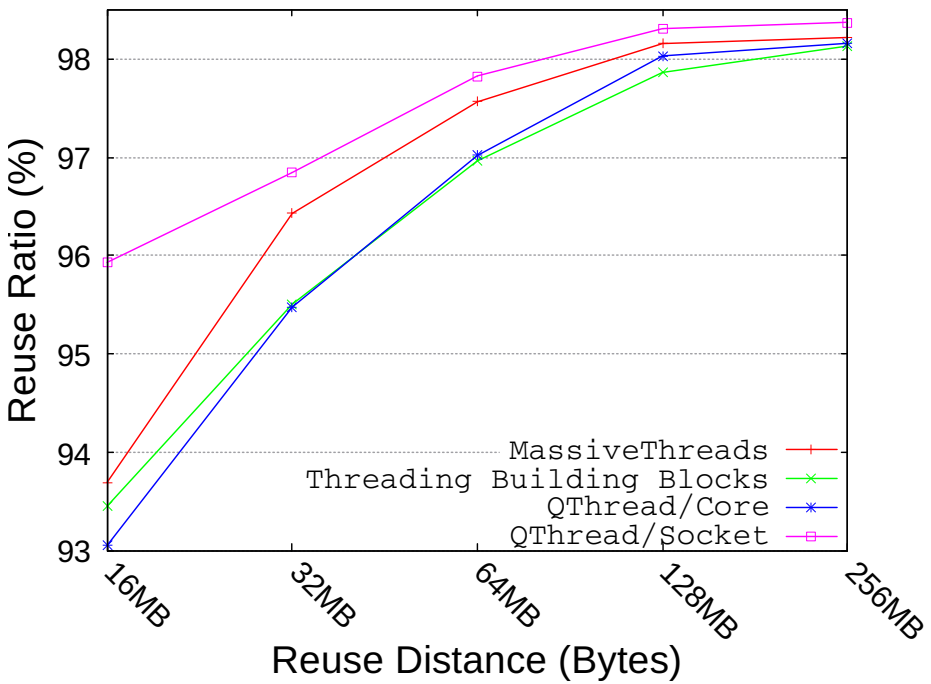


KRD histograms and performance

- Want to understand how the KRD metric correlates with hardware performance metrics
- We choose a multisocket low overheads scenario: Matmul on 2 sockets / 12 cores
  - **Low Overhead**: MTH, TBB, QTH/Core overheads around 1.1-1.2×
  - **Moderate Overhead**: QTH/Socket overhead 1.35×

Hardware Metrics and KRD for Matmul on 2 sockets

Runtime	Exec. Time	OVR_{12}	LLC Misses	Kernel Time & Inflation
MTH	1.642 sec	1.094×	1.829×10^6	17441ns (1.0250×)
TBB	1.742 sec	1.11×	2.807×10^6	17898ns (1.0519×)
QTH/Core	1.859 sec	1.21×	2.339×10^6	17767ns (1.0441×)
QTH/Socket	2.111 sec	1.34×	1.987×10^6	18401ns (1.0814×)



- LLC misses correlate to data reuse histograms
- QTH/Socket LLC miss rate and WTI too large.
- Runtime activity is causing additional misses and contention on memory subsystem

Table of Contents

Discussion

Tracks only global data accesses in bulk

- The user needs to ensure that stack accesses are negligible
- For matmul we found that stack accesses are less than 1%

No modeling of the effects of cache coherence

- Affects multisocket scenario, our results are optimstic

No measurement of spatial locality

- A spatial locality metric could be added to model prefetchers.

Table of Contents

Summary of findings

- We developed a tool based on the reuse distance to study reuse in task parallel applications. The tools is designed to be lightweight and can provide fast comparison of different implementations.
- Although the tool was developed to compare task parallel runtimes, it can be applied to any shared memory model.
- Our experiments indicate that the reuse distance histograms correlate with scheduler policies and with hardware metrics

Thank you!

Questions?